Putting the Pieces Together: The Construction of a Reliable Virtualizing Object-Based Storage Stack

David C. van Moolenbroek, Raja Appuswamy, Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit Amsterdam
{*david, raja, ast*}*@minix3.org*

***Abstract*—The operating system storage stack is an important software component, but it faces several reliability threats. The research community has come up with many solutions to address individual parts of this reliability problem. However, when considering the bigger picture of constructing a highly reliable storage stack out of these individual solutions, new questions arise regarding the feasibility, complexity, reliability, and performance of such a combination. In previous works, we have designed a new storage stack called Loris, and developed several individual reliability improvements for it. In this work, we investigate the integration of these improvements with each other and with new functionality, in two steps. First, we add new virtualization extensions to Loris which challenge assumptions we made in our previous work on reliability. Second, we combine all our extensions to form a reliable, virtualizing storage stack. We evaluate the resulting stack in terms of performance and reliability.**

***Keywords*-Operating systems, File systems, Platform virtualization, Software reliability, Fault tolerance**

I. Introduction

All computer systems are vulnerable to various faults, originating from both software and hardware. Such faults have the potential to completely subvert the operation of the system. Given that prevention and hardware protection are often infeasible or too costly, reliability can be improved using software-based isolation, detection, and recovery techniques. The storage component of the operating system is of particular interest in this regard: it is relied upon by all applications, it is highly complex, and it uses several resources which may fail in various ways. Since applications expect the storage stack to operate perfectly, unhandled failures in this stack may result in loss of user data.

The storage stack faces several important reliability threats. The most well-known threat is a *whole-system failure*, where the entire system goes down unexpectedly. The underlying storage hardware may experience a *storage device failure*, which may range from fail-stop failures to various forms of silent data corruption. Since the storage stack typically uses a large amount of available RAM for caching purposes, it is also vulnerable to *memory corruption* from sources such as cosmic rays. Finally, the complexity of the storage stack makes it susceptible to failures due to *software bugs*.

As a result, there has been much research on improving the reliability of the storage stack in various ways (e.g., [1]–[12]). Such research typically focuses on a single reliability threat, limits itself to a single part of the storage stack, and often makes various assumptions about the component being protected. The result is extra reliability with low runtime overhead but with a limited scope. As such, while these efforts provide essential pieces towards making the storage stack more reliable, a top-down look at constructing a highly reliable storage stack raises new, important questions:

- Can a combination of individual techniques provide significant reliability coverage of the entire stack?
- How do these techniques hold up in the light of new functionality that may violate previous assumptions?
- What advantages, limitations, and complexity result from integrating various pieces with each other?
- What is the performance and reliability of such a combination of individual low-overhead techniques?

We believe that we are in a good position to provide initial answers to these questions. We have previously designed a storage stack called Loris, which is layered in a more modular way than the traditional storage stack [9]. In subsequent work, we have looked at the aforementioned reliability threats by considering individual layers of the stack in isolation [10]–[12]. By exploiting the specifics of the design of these layers, we were able to provide individual solutions that provide strong reliability and add little overhead.

In this paper, we attempt to answer the above questions, in two steps. In recent work, we have presented a new approach to virtualization [13]; our first step in this paper consists of implementing support for this form of virtualization to Loris. In particular, we extend Loris with object virtualization, copy-on-write, and deduplication functionality, thereby forming **vLoris**. In addition to its intrinsic value, this first step serves two purposes. First, it improves reliability by placing the upper layers of the storage stack in their own failure domain. Second, the new functionality provides a good test case for the assumptions in our earlier work on reliability.

In the second step, we combine our earlier individual works on reliability with each other and with the new virtualization support, thus forming a virtualizing storage stack which is at least as reliable as the sum of the individual pieces. This new version of Loris, **rvLoris**, provides various levels of robustness against the four mentioned reliability

threats across all its layers. However, given that we made strong assumptions about the layers in previous work, this integration effort required resolving several new issues. We describe the resulting changes that we had to make. Finally, we provide answers to the posed questions by evaluating the design, complexity, performance, and reliability of rvLoris.

The rest of the paper is laid out as follows. Sec. II provides a necessarily elaborate overview of our relevant previous work on Loris, reliability, and virtualization. In Sec. III, we describe the virtualization extensions that make up vLoris. In Sec. IV, we describe our efforts to integrate all reliability and virtualization features to form rvLoris. In Sec. V, we evaluate the performance and reliability of all our changes. Sec. VI describes related work, and Sec. VII concludes.

II. Background

In this section, we describe the prior work on which this paper builds: the Loris storage stack (Sec. II-A), our reliability extensions to it (Sec. II-B to II-D), and the role of Loris in our new approach to virtualization (Sec. II-E).

A. The Loris storage stack

The basis of all our work is a new object-based storage stack called Loris [9]. This stack replaces the file system and software RAID layers of the traditional storage stack (Fig. 1a) with four new layers (Fig. 1b). These four layers communicate among each other in terms of *objects*: storage containers which are made up of a unique identifier, a variable amount of byte data, and a set of associated attributes. Each of the three lower layers exposes the following object operations to the layer above it: `create`, `delete`, `read`, `write`, `truncate`, `getattr`, `setattr`. In addition, a `sync` operation flushes all dirty state. Loris offers advantages in the areas of reliability, flexibility, and heterogeneity [9].

At the bottom, the **physical** layer manages the layout of underlying storage devices, exposing a *physical object* abstraction to the layers above. The physical layer consists of one or more *modules*, each managing the layout of a single underlying device using a layout suitable for that device type. The physical modules are required to support parental checksumming to detect all forms of disk corruption [9], [14]. Our initial physical module, PhysFS, implements a layout based on the traditional UNIX file system, using *inodes* to store the objects with their attributes and data.

The **logical** layer adds support for RAID-like per-object redundancy. It exposes a *logical object* abstraction to the layers above. Each logical object has an individual RAID-like policy and is made up of one or more objects on different physical modules. For example, an object may be mirrored or striped across some or all devices. The logical layer maintains a mapping from each logical object to its policy and physical objects. It stores the mapping in a *metadata object*: an object which contains storage stack metadata and is mirrored across all devices. When a physical module reports a checksumming error, the logical layer uses the per-object redundancy to restore a known-good copy of the object. In the absence of redundancy, it still prevents propagation of corrupted data.

The **cache** layer uses available system memory to cache pages of object data. It also caches object attributes, and defers flushing dirty pages and object `create` and `setattr` operations for improved performance. Due to its limited role, the cache is by far the least complex layer of the stack.

Together, these lower three layers effectively implement an object store. On top of these layers, the **naming** layer constructs a POSIX file system hierarchy out of the loose objects. It processes requests from the Virtual File System layer (VFS) and implements support for file naming, directories, and POSIX attributes. It stores files and directories as objects, and POSIX attributes as object attributes.

We have implemented the Loris storage stack in the MINIX 3 microkernel operating system [15]. As a result, all layers and modules of the storage stack are isolated user-mode processes, limited in their interprocess communication (IPC) in accordance with the principle of least authority. MINIX 3 has facilities to detect crashes in user-mode operating system processes and to start a clean copy of a failing process, although recovery of both state and ongoing requests is left to the process itself. The infrastructure by itself is sufficient to recover from failures in device drivers [16].

Summarizing, in terms of reliability, Loris can detect and recover from storage device failures by design, and MINIX 3 provides the necessary (but not sufficient) infrastructure to detect and recover from software failures ("crashes"). In subsequent, separate projects, we have further improved the robustness of Loris, and we describe these projects next.

B. Improved reliability in the physical and logical layers

In previous work [10], we argue that the storage stack should provide a unified infrastructure to recover both from whole-system failures and from crashes in the logical and physical module processes. For both failure types, recovery

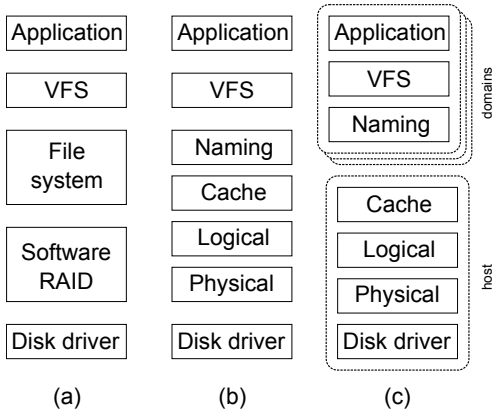

Figure 1. A schematic diagram of the layers of (a) the traditional storage stack, (b) the Loris storage stack, and (c) the Loris stack with virtualization.

relies on the creation of consistent on-disk *recovery points*. A recovery point is created with the `sync` operation, which goes down the entire stack, during which all layers flush their dirty state to the layers below. Finally, at the physical layer, all modules create a recovery point on their devices. As proof of concept, we developed a new physical module, TwinFS, which supports such on-disk recovery points.

This infrastructure, when combined with cross-device data synchronization, guarantees recovery to the last recovery point after a whole-system failure. The same infrastructure allows for recovery of a crash in the lower two layers, when combined with an in-memory log in the cache layer.

The in-memory log records all modifying operations sent down to the lower layers since the last recovery point was created. It is cleared after a `sync` call. If one of the processes in the logical or physical layer crashes, all modules in this layer are restarted, and thus together restore to the last recovery point. After that, the cache layer replays the log, thereby bringing back the lower layers to the latest state. This approach provides transparent recovery with strong guarantees from transient crashes in the lower layers, and automatically covers any new features in these layers as well.

We exploit two properties of object storage to reduce logging complexity and overhead. First, since all objects are independent from each other, the log need not keep track of the order of operations between different objects. Second, operations that are later obsoleted (e.g., a `write` followed by a `truncate`) can be eliminated from the log, so that the operations need not be replayed in chronological order.

Our evaluation shows that TwinFS adds a performance overhead of around 5–8 %. The log adds little performance overhead, but does require a large amount of extra memory, mainly to log pages that have been flushed and then evicted from the cache layer's (separate) main memory pool.

C. Improved reliability in the cache layer

As indicated, the Loris physical layer generates and verifies checksums of data and metadata blocks in order to detect disk corruption. We argue that checksums for data blocks should be propagated from and to the cache layer [11]. With such a facility in place, the cache layer can then use the checksums for two reliability purposes: detecting memory corruption and recovering itself from a crash.

First, the cache layer uses a large amount of system memory for data caching purposes. Therefore, there is a substantial chance that if a random DRAM bit flip occurs, it will affect a page in the cache. In order to detect such memory corruption before it is propagated, we change the cache layer to use the checksums for verification of pages right before application read calls. Recovery is often possible as well: corrupted clean pages can be restored from disk.

Second, crash recovery of the cache layer is difficult because this layer typically contains a substantial amount of state which cannot be recovered from elsewhere: delayed page writes, and `create` and `setattr` operations. Thus, if the cache process crashes, such dirty state must be recovered from the memory of the crashed instance of the process.

In order to ensure that the dirty state in the cache has not been corrupted as part of the crash, we change the cache layer to checksum the necessary data structures during runtime and verify them upon recovery. The state includes dirty pages; thus, each page is now checksummed right after being modified instead of when it is flushed to disk. All dirty state is tracked using a set of self-checksumming data structures, together called the Dirty State Store (DSS). During normal cache operation, the DSS forms a fully checksummed tree of all dirty state within the cache's memory. If the cache crashes and its recovery procedure detects a checksum mismatch in the DSS tree, the cache is not recovered, thus minimizing the risk that corrupted data reaches applications. An update of a page and its DSS checksum is not atomic; thus, this technique can recover from most but not all fail-stop failures.

For application-transparent crash recovery, the system must also deal with operations that were being processed by the cache at the time of the crash (as per Sec. II-A). We require that either the naming layer or the low-level IPC system repeat pending operations after a restart of the cache. Since all Loris operations are idempotent, their repetition ensures a correct outcome even in the light of earlier partial processing.

The paper shows that the integration of the two techniques further strengthens both, and that the combination of hardware-implemented checksumming and simple techniques can keep performance overhead down to around 1 % while providing reasonable detection and recovery guarantees.

D. Improved reliability in the naming layer

While the naming layer may be highly complex, at its core it simply translates each POSIX request from VFS into a number of Loris object operations to the cache layer. We exploit this fact to implement transparent recovery from naming-layer crashes [12], by making the layer stateless.

We change the naming layer to group all modifying operations spawned by a single VFS request into a transaction, which is then sent to the cache layer at the end of the request. The cache layer is responsible for processing each transaction as an atomic unit. As a result, the lower layers always contain a consistent state. If the naming process crashes, it will therefore also reload a consistent state upon recovery.

That leaves any VFS requests that were previously in flight. VFS reissues all those requests after a naming-layer restart; during runtime, the naming layer uses the transaction system to write markers to special files in order to determine after a crash which VFS requests were already processed before.

The cache layer places strict rules on the operations that may be part of a single transaction. For example, a transaction may contain only one `delete` operation: this operation can not be rolled back and must therefore be processed last. The cache further exploits the high-level nature of these

transactions to limit rollback records to a minimum. In addition, the grouping of operations saves on low-level IPC; the resulting performance overhead is between −1 and 2 %.

E. A new approach to virtualization

In our most recent work [13], we present a new virtualization alternative which combines several good properties of virtual machines and operating system containers. The boundary between the virtualizing (*host*) and the virtualized (*domain*) parts of the system is placed such that the host can share and globally optimize resources across domains, while each domain has the freedom to implement a *system layer* which exposes any desired abstractions to its applications. As a result, our alternative is more lightweight than virtual machines and more flexible than operating system containers.

A key part of the design is a new form of storage virtualization: the host exposes an object store to the domains, and the system layer of each domain can use this object store to construct any file system abstraction as it sees fit for its applications. The central object store provides each domain with a private namespace of object identifiers; storage changes from one domain are never visible to another domain. However, the object store is free to share resources between domains, by mapping multiple domain-local objects to a single underlying global object using copy-on-write (CoW) semantics. Furthermore, if the object store implements a page cache, such global objects can be shared not only on disk but also in memory. Since many domains are expected to use the same files in practice, this results in more efficient storage and memory usage in the common case.

The structure of Loris makes it a suitable basis to construct such a virtualizing object store. To this end, the storage stack is split in two (Fig. 1c). The cache, logical, physical, and driver layers become part of the host, together implementing the centralized object store. The VFS and naming layers become part of the system layer in the individual domains. Thus, each domain is expected to have its own instance of the naming layer, and all these naming modules make "host calls" to the cache layer to perform operations. All the storage stack modules remain separate processes, and the MINIX 3 microkernel implements domains by virtualizing IPC between groups of processes. As a result, the cache layer now receives a *domain identifier* along with each incoming operation. With this as a given, the cache layer must implement support for storage virtualization, which is the subject of the next section.

III. vLORIS: SUPPORT FOR VIRTUALIZATION

In this section, we describe the design and implementation of our new storage virtualization extensions to Loris. We call the result vLoris. We add object virtualization and copy-on-write support to the cache layer (Sec. III-A). We reuse our previous work on transactions to support whole-system failure recovery (Sec. III-B). We show that effective copy-on-write support requires attribute management to be moved into the cache layer (Sec. III-C). Finally, we add object-level deduplication to the cache layer (Sec. III-D).

A. Object virtualization and copy-on-write

We start by modifying the Loris cache layer to support object virtualization and object-granular copy-on-write. To this end, the cache layer maintains for each domain a *domain mapping object*: a metadata object (Sec. II-A) which stores the mapping from the domain's local object identifiers to global object identifiers. When the cache gets a call from a naming module, it translates object identifiers in the call using the mapping for the naming module's owning domain. Thus, each domain can access only the objects in its mapping.

In order to support copy-on-write, we change the cache layer to keep a reference count for all global objects. These reference counts are stored in an additional metadata object. We use the domain mapping and reference count metadata objects to implement well-known copy-on-write semantics for the `create` and `delete` operations.

Whenever a domain modifies an object with a reference count greater than one, the cache layer makes a copy of the global object. In order to copy objects efficiently, we introduce a new `copy` operation which is implemented in the logical and physical layers and used by the cache layer. Currently, the logical layer forwards the call to the appropriate physical modules, which make a full copy of the underlying object. A future implementation of this operation could implement copy-on-write support at a subobject granularity.

B. Transactions

An important part of virtualization is defining the interface between the host and the domains. In order to allow vLoris to recover from whole-system failures, its host/domain interface has to be changed, and thus, we consider this reliability aspect to be a part of the virtualization changes.

As described in Sec. II-B, in order to establish a recovery point, all the Loris layers have to flush down their dirty state as part of a stack-wide `sync` call. This includes the naming layer. However, the naming layer is part of the domains, and the domains are untrusted from the point of view of the host system. Thus, they cannot be relied upon to cooperate in creating recovery points, since upcalls from the host system into the domains would introduce a dependency on untrusted components. Even though a naming module could only create inconsistencies for itself, such upcalls would at least need a timeout, and it would be very difficult to choose a reasonable value for this timeout. Avoiding upcalls is thus preferable.

We solve the problem by adopting the naming-layer transaction support described in Sec. II-D. As a result, the naming layers need no longer be involved in establishing a recovery point at all: the transactions ensure that the cache layer always has a consistent state, and thus, the host system can create a new recovery point at any time and without the involvement of the domains. As a side benefit, this merges in most of the support for crash recovery of naming modules.

C. Attribute localization

In our work on naming-layer transactions, we disabled updating file access times for performance reasons. For this work, we added support for *lazy* access time updates in the naming module implementation. The naming module gathers access time updates for files, and periodically sends these down to the cache layer. While the naming layer is thus no longer stateless, deferring such updates does not introduce consistency violations. At most, a whole-system or process failure will cause some access time updates to get lost.

However, since each object is stored together with its attributes in the physical layer, the cache layer must copy any CoW-shared object whenever its attributes are changed. Access time updates make this design untenable: any read call on a file now causes sharing to be broken for that file.

This case makes clear that in general, effective storage sharing at object granularity requires that such sharing apply to the object's contents only, and not its attributes. Thus, in our stack, we can no longer let the physical layer manage attributes. Instead, we "localize" attributes by storing them in the local mapping entries of the domain mapping objects, along with each global object identifier. The `getattr` and `setattr` operations are thus handled by the cache layer, and `setattr` no longer causes objects to be copied. The physical layer no longer stores attributes in its inodes.

D. Object-level deduplication

So far, the only way to increase the reference count of an existing object is to load a mapping for a new domain. Thus, after creation, a domain's storage resources will only ever diverge from the shared pool. There are however many scenarios in which multiple domains end up with the same objects (long) after the domains' creation, for example as a result of a software update being installed in several domains.

Thus, storage resources can be saved with object-level deduplication: a facility that finds and merges global objects with the same content and establishes copy-on-write mappings to the merged objects. We believe that deduplication at the object level is an acceptable tradeoff between yield and overhead [17], although as before, our object store can later be extended to implement subobject deduplication with no interface changes exposed to domains.

We implement support for such deduplication in the cache layer, since the object copy-on-write facility is implemented there as well. As we will now show, we use a weak (non-cryptographic) form of object content hashing to generate deduplication candidates with good accuracy and little overhead. We perform background content comparison on the candidate objects to achieve eventually-perfect deduplication.

As described in Sec. II-A, Loris implements parental checksumming of blocks in the physical layer. We reuse these checksums to generate object hashes: the hash of an object is the exclusive-OR (XOR) of all its block checksums. The cache layer maintains an *index* which maps between global objects and their hashes. This index is stored in a metadata object, but also kept entirely in memory if possible.

Whenever the cache sends down an operation that changes the hash of an existing object–that is, a `write` or a `truncate` operation–the lower layers reply with a value representing the XOR of all changes to the object's block checksums. Since the physical modules visit all involved block pointers as part of the operation, getting the old block checksums from there is cheap. The cache XORs the reply value with the old object hash to obtain the new object hash, and modifies the index accordingly. While inserting the new hash value into the index, the cache checks whether any other objects have the same hash value, and if so, it registers the changed object as a candidate for deduplication.

At convenient times, the cache layer goes through the candidates list and compares each candidate against its potential matches, by doing a full comparison of the contents of the objects. We implemented a simple approach of reading and comparing potential matches at timed intervals. However, many optimizations are possible here: comparing candidates before their pages are evicted, comparing block checksums before comparing actual contents, testing candidates only during times of low I/O activity, etcetera.

The cache layer has no reverse mapping from global to domain-local objects. Therefore, the candidates list contains pairs of domain identifiers and domain-local object identifiers. Upon actual deduplication, the candidate's domain mapping is changed to point to its match. The lack of a reverse mapping also implies that deduplication of objects within a single domain is automatic and cannot be prevented.

IV. rvLoris: integration of reliability support

In the previous section, we already merged in support for naming-layer transactions. We now describe the changes necessary to merge our other reliability extensions with the new virtualization support as well as each other. Our goal is to maintain the same reliability guarantees as provided by the individual works. Since all the functionality comes together in the cache layer, all the new changes involve this layer in particular. We describe the changes that allow lower-layer restarts to work with virtualization (Sec. IV-A), and cache restarts to work with transactions (Sec. IV-B), object virtualization (Sec. IV-C), and object deduplication (Sec. IV-D). Finally, we discuss the results (Sec. IV-E).

A. Lower-layer restarts versus virtualization

In order to support lower-layer crash recovery for the logical and physical layers (Sec. II-B) in the presence of the new virtualization, we have to make two modifications to the way the cache layer manages its in-memory log.

First, the logging facility was previously a separate submodule within the cache implementation: it maintained its own data structures and allocated its own memory to store logged data. This strict separation helped us in gathering research

data about its implementation complexity and memory usage. However, the separate memory allocation was also a necessity: the only way to clear the log is to create a new recovery point, and since the cache layer itself could not initiate the creation of such points due to dirty state in the naming layer, the log was never allowed to run out of memory.

However, a real-world storage stack does not have infinite memory at its disposal; more realistically, such a log makes use of the same memory as the main cache [6]. Since the naming-layer transactions now allow the cache layer to create recovery points at any time (as per Sec. III-B), there is no longer a need to allocate separate memory. Thus, we change our logging implementation to use the cache's main memory pool. This in fact simplifies our logging implementation: by using the main object and page data structures, the logging submodule no longer needs to manage duplicate data structures. In the new situation, cached pages that are in use by the log may not be evicted until the next `sync`, but they may be freely read, modified (with new incoming `write` operations), and discarded (with `truncate` or `delete`).

Second, the addition of the new `copy` operation to the lower layers requires that the cache log include this operation as well. However, the `copy` operation violates the two assumptions that we previously relied on to keep the log simple: full independence between objects, and the ability to throw out any logged operations that have become obsolete. After all, the result of the copy depends on the state of the source object, and thus, a post-crash replay of this operation requires that the source object be restored in the same state first. The log must take this into account; e.g., an object `truncate` may not discard pages logged for an earlier `write` if the object was subject to a `copy` in the meantime.

The simplest solution is to establish a new on-disk recovery point after every `copy` call, thereby clearing the log. However, a system that hosts many domains can expect frequent copying, making this an expensive solution. Instead, we expand the cache logging facility with a partial ordering system, using hidden object and page data structures to log the precopy object state needed for replay. Basic dependency tracking allows log replay in a correct order: when a `copy` operation is logged, new data structures are created for the source object, and the old source object data structures are hidden and marked as dependency for the new ones.

B. Cache restarts versus transactions

As we have shown, the transactions system is used for both naming-layer reliability (Sec. II-D) and, indirectly, for whole-system failure recovery for virtualization (Sec. III-B). However, the addition of transaction processing to the cache layer has implications for the cache layer's crash recovery.

In our earlier work (Sec. II-C), we required that pending operations be repeated after a cache-layer restart, and relied on idempotence to correct earlier partial completion of those operations. This no longer works with transactions. A single transaction may spawn multiple modifying operations to the lower layers, each of which may fail, in which case the entire transaction must be aborted. If some of these operations are issued successfully and then the cache crashes, and the transaction ends up being aborted upon repetition, the subsequent rollback will revert to an inconsistent state.

We solve this issue by adding rollback records for active transactions to the Dirty State Store (DSS). When the cache processes a transaction, it first creates a rollback record in a known (per-thread) location, along with a checksum generated of its contents. When the cache completes processing the transaction, it discards the record using a (checksummed) special value. After a crash in the cache, the cache's recovery procedure first verifies the checksums of all rollback records in the old memory image, and proceeds with recovery only if all are valid. It then issues a rollback for all those records, thus restoring the cache to a consistent state.

C. Cache restarts versus object virtualization

The cache layer's object virtualization adds a number of extra, risky steps for operation processing. Object creation, copying, and deletion actions all require changes to multiple objects: a domain mapping, the global reference counting metadata object, and often the global target object itself. With these extra steps, partial completion of even a single operation is no longer recoverable by repetition, as nonatomic metadata updates likely result in fatal inconsistency.

In order to ensure that the cache layer can still either recover correctly or detect internal corruption, we integrate the object virtualization steps into our transaction system. We split each of the creation, copying, and deletion actions into a prepare routine and a commit routine. We create a rollback routine for each action, which restores the old state regardless of any changes made in the meantime. We add new checks to ensure isolation between transactions, for example with respect to allocation of global IDs. Also, since it is common that two operations modify the same object within the same transaction, the prepare phase has to consider any actions already scheduled within the same transaction.

However, the cache cannot roll back `delete` operations already sent to lower layers. Thus, after a crash of the cache, a rollback of a delete action is not always possible. Therefore, we have to make one exception in the cache's internal consistency model: a domain-local object ID may temporarily point to a global object which has a nonzero reference count but does not exist in the lower layers. Since pending requests must be repeated after a cache restart (Sec. II-C), such inconsistencies are always corrected quickly.

D. Cache restarts versus object deduplication

The addition of object deduplication to the cache layer requires three more extensions to its crash recovery system.

First, when the deduplication facility finds that two objects are identical, they are merged. Like the create, copy, and

delete actions, such a merge action requires an atomic set of changes, and thus needs to be wrapped in a (purely cache-local) transaction. The action includes a `delete` on one of the two objects, making it similar to a delete action. However, a cache crash must not introduce a similar inconsistency for a merge action: the action was not started by a call from the naming layer, and thus may not be temporary. We therefore opt to let the merge rollback routine perform a roll-*forward* on the merge action. Thus, upon crash recovery, the merge will always be finished, thereby preventing inconsistency.

Second, the cache updates the deduplication index with a new checksum based on the reply to a `write` or `truncate` call down to the lower layers. A crash may cause that reply to get lost, in which case the deduplication index becomes desynchronized. We solve this by adding an index invalidation bitmap to the DSS. Each set bit invalidates a chunk of the index; a bit is set while any index entry (i.e., object) in that chunk has a pending downcall. Upon recovery, the cache must reconstruct all entries in the marked chunks. It does so by explicitly requesting the XOR'ed checksum from the lower layers for each affected object, using `getattr` calls.

Third, in order to ensure eventually-perfect deduplication even in the presence of cache crashes, the duplication candidates list must be preserved across restarts. Thus, it would have to be tracked by the DSS, which would be simple but costly. We have not yet added such recovery support.

E. Discussion

We now turn to the questions posed in the introduction. First of all, we have shown that it is indeed feasible to construct a storage stack with advanced functionality and a strong focus on reliability, by combining several partial solutions. Even though the integration forced us to depart from several simplifying assumptions made in earlier work, the subsequent changes have not compromised any of the original techniques. In addition, this study further strengthens our notion that when considering the bigger picture, individual building blocks such as checksums, recovery points, and transactions can be exploited for multiple reliability purposes.

Our resulting stack has several limitations. For example, we currently assume that software crashes of the individual layers are independent. Therefore, we deliberately choose not to protect the logging data structures with the DSS in the cache, since this would introduce more complexity for the unlikely event of crashes of multiple layers at once. However, since failures may propagate to other layers, the assumption may not hold in practice. Thus, the stack may benefit from more thorough checks at the layer boundaries (along the lines of Recon [8]). As another example, the only software bug protection we provide for the VFS layer is the isolation inherent to virtualization: instead of taking down the entire system, a VFS crash will now take down only its containing domain. VFS maintains large amounts of application state, making it an interesting target for future research.

It is clear that our integration adds more complexity, mainly to retain crash recovery support for the cache layer. Thus, alternative approaches should be considered for that aspect. It would be possible to place the virtualization and deduplication functionality in a new layer above the cache. However, this would not simplify the problem: in order to support recovery from a crash, the new layer would have to be stateless, modifying the transactions it forwards on the fly. That is effectively what our changes to the cache layer do. Thus, aside from the performance overhead of an extra layer in the critical path, such a new layer would serve to make our implementation cleaner, not less complex. Instead, we believe that we could reduce the added complexity by using generic post-crash rollback functionality, for example based on work by Vogt et al [18]. In order to keep runtime overhead low, this technique would require integration with our high-level transaction optimizations; that is part of future work.

V. Evaluation

In this section, we provide further answers to our questions by evaluating our work. We first evaluate the performance of vLoris (Sec. V-A). We then continue with rvLoris, measuring its performance (Sec. V-B) and reliability (Sec. V-C).

A. vLoris performance

We use storage macrobenchmarks to evaluate the performance impact of our virtualization changes to the storage stack. In all performance experiments, we use an Intel Core 2 Duo E8600 PC with 4 GB of RAM and a 500 GB 7,200 RPM Western Digital Caviar Blue SATA test disk, and a version of MINIX 3 that implements our new virtualization support. The benchmarks are run from a single domain. We configure the Loris naming layer to enable access time updates, limit the Loris cache layer to 1 GB of memory, and use PhysFS in the Loris physical layer, operating on the first 32 GB of the disk. The system performs a `sync` once every five seconds and whenever 10 % of the page cache has become dirty.

We use the following benchmarks and configurations: an OpenSSH unpack-and-build test which unpacks, configures, and compiles OpenSSH in a `chroot` environment; a source compilation of MINIX 3 in a `chroot` environment; PostMark, with 800K transactions on 40 K files across 10 directories, using 4 to 28 KB file sizes and 512-byte unbuffered I/O operations; FileBench File Server, single-threaded, run for 30 minutes at once; and, FileBench Web Server, single-threaded, changed to access files according to a Zipf distribution ($\alpha = 0.98$), run for 30 minutes as well. Most of the benchmarks run (almost) entirely in memory, which is the worst case for most of our changes.

We run the benchmarks on several Loris versions along its transition into vLoris, in the order as described in Sec. III. With physical-layer checksumming turned off, we first run the original Loris (*Baseline*), to which we incrementally add object virtualization and copy-on-write support (*Virtualization*),

Table I
Macrobenchmark performance of vLoris. The OpenSSH and MINIX 3 build results are in seconds (lower is better). The PostMark results are in transactions per second (higher is better). The File Server and Web Server results are in operations per second (higher is better).

Benchmark	Baseline	Virtualization	Transactions	Attributes	Checksums	Index	Compare	Merge
OpenSSH build	597 (1.00)	600 (1.01)	616 (1.03)	622 (1.04)	617 (1.03)	614 (1.03)	623 (1.04)	629 (1.05)
MINIX 3 build	824 (1.00)	833 (1.01)	842 (1.02)	847 (1.03)	843 (1.02)	842 (1.02)	851 (1.03)	843 (1.02)
PostMark	258 (1.00)	255 (0.99)	245 (0.95)	243 (0.94)	243 (0.94)	243 (0.94)	244 (0.95)	183 (0.71)
File Server	2423 (1.00)	2421 (1.00)	2541 (1.05)	2499 (1.03)	2489 (1.03)	2463 (1.02)	450 (0.19)	324 (0.13)
Web Server	17731 (1.00)	17537 (0.99)	17479 (0.99)	17493 (0.99)	17474 (0.99)	17534 (0.99)	17520 (0.99)	17631 (0.99)

support for naming-layer transactions (*Transactions*), and attribute localization (*Attributes*). On top of these changes, we turn on checksumming with Fletcher's checksum algorithm (*Checksums*), and add deduplication support in three steps: the maintenance of the in-memory deduplication index and the candidates list (*Index*), comparison of candidates (*Compare*), and actual deduplication (*Merge*).

We run each combination of benchmark and Loris configuration at least five times and report the average. The results are shown in Table I. The numbers in parentheses represent performance relative to the baseline. For the OpenSSH and MINIX 3 build benchmarks, lower is better. For PostMark and the two FileBench benchmarks, higher is better.

Since the benchmarks are run from a single domain, the small overhead of object virtualization and copy-on-write support (*Virtualization*) is due entirely to record keeping of the domain mappings and reference counts.

The addition of the transactions support (*Transactions*) has mixed effects. In particular, PostMark takes a hit while File Server speeds up. Compared to our earlier evaluation [12], our new PostMark configuration performs subpage writes, thereby exposing an implementation-specific overhead of such writes: when wrapped in transactions, these writes require an extra kernel call. We are working on resolving this issue.

The File Server speedup is a result of the change in how access times are updated in the naming layer, which in turn affects object caching behavior in the cache layer. This result is specific to the combination of File Server's access patterns and our cache size, and does not extend to other benchmarks.

The localization of attributes (*Attributes*) has a small effect on performance. In the physical layer, the attributes are now stored as data and no longer part of inodes, which could have a negative effect on locality. However, the cache layer colocates the attributes with their corresponding domain mapping entries, thus achieving another form of locality.

The *Checksums* column shows that enabling Fletcher checksumming–as needed for deduplication–adds practically no overhead, although this can be expected with benchmark configurations that mainly stress the cache. The *Index* column shows that maintaining the index and generating candidates has no further impact on performance, thus confirming that the inline part of our deduplication system has little overhead.

When comparing candidates against potential matches (*Compare*), File Server shows significant overhead. During runtime, PostMark and File Server both generate new files that have all-zeroes contents. Thus, for both benchmarks, all same-sized files end up being flagged as candidates for deduplication. For PostMark, the entire workload fits in the page cache, and thus, the contents comparison operates on cached pages only. For File Server however, the candidate comparison thrashes the cache by reading the matches from disk. Sec. III-D described several possible improvements.

The last column (*Merge*) shows that the same two benchmarks suffer even more from actual deduplication. Merging objects is cheap, but a large number of merged objects end up being appended to later, forcing the storage stack to copy the underlying object again.

In general, it can be expected that making copies of entire objects is expensive. As we mentioned before, vLoris could be extended with subobject (block-level) copy-on-write and deduplication support. In this work, we have presented the necessary infrastructure to support object-level virtualization in Loris, and if we discount the deduplication comparison and merging, our changes impose between −2 and 6 % performance overhead for the workloads in our tests.

B. rvLoris performance

For rvLoris, we start with the vLoris *Checksums* version, but we switch from PhysFS to TwinFS, with a 16-block twin offset (*TwinFS*). In terms of reliability, this configuration includes support for disk corruption detection and whole-system failure recovery. On top of the *TwinFS* configuration, we measure the individual performance of the following extensions: crash recovery of the naming layer (*CR-Naming*), the cache layer (*CR-Cache*), and the lower layers (*CR-Lower*); detection of memory corruption in the cache layer (*MD-Cache*); and, again, maintenance of the deduplication index and candidates list (*Index′*). Finally, we enable all extensions at once (*rvLoris*). The results are shown in Table II, including numbers relative to the original *Baseline* from Table I.

When compared to the earlier *Checksums* results, the *TwinFS* results show a 1–7 % overhead. While no worse than our original results (Sec. II-B), this is substantial. Other on-disk consistency formats may be able to reduce the overhead.

Since vLoris already includes transaction support, crash recovery of the naming layer (*CR-Naming*) only adds writing markers to avoid repeating requests after recovery (Sec. II-D). Since these write operations can always be combined with an existing transaction, this feature adds very little overhead.

Table II
Macrobenchmark performance of rvLoris.

Benchmark	TwinFS	CR-Naming	CR-Cache	CR-Lower	MD-Cache	Index′	rvLoris
OpenSSH build	657 (1.10)	650 (1.09)	657 (1.10)	656 (1.10)	659 (1.10)	656 (1.10)	663 (1.11)
MINIX 3 build	890 (1.08)	887 (1.08)	888 (1.08)	892 (1.08)	913 (1.11)	900 (1.09)	917 (1.11)
PostMark	235 (0.91)	235 (0.91)	229 (0.89)	236 (0.92)	230 (0.89)	235 (0.91)	226 (0.88)
File Server	2332 (0.96)	2335 (0.96)	2299 (0.95)	2322 (0.96)	2299 (0.95)	2295 (0.95)	2269 (0.94)
Web Server	17297 (0.98)	17325 (0.98)	17313 (0.98)	17441 (0.98)	16979 (0.96)	17431 (0.98)	16867 (0.95)

The new cache crash recovery (*CR-Cache*) initially had high overheads, caused by extra checksumming: every small change to a metadata object page (e.g., changing a reference count) requires immediate full-page rechecksumming for the DSS. We implemented subpage checksum update support for Fletcher, which reduced the overheads by a large margin. The remaining overhead is still due to the cost of Fletcher checksumming in software; in our previous experiments (Sec. II-C), we used checksumming support in hardware.

The cache-layer logging for lower-layer restarts (*CR-Lower*) has little overhead. The sync policy which keeps the system responsive by not allowing the cache to build up too many dirty pages, also prevents that flushed dirty pages kept around for the log put a strain on the cache.

The memory corruption detection (*MD-Cache*) overheads are due entirely to checksumming, like with *CR-Cache*. Deduplication indexing and candidate generation (*Index′*) yields overheads similar to those of the earlier runs (*Index*).

Given that *rvLoris* combines all extensions, it is not surprising that it performs slightly worse than the worst-performing extension. Compared to the *TwinFS* results, rvLoris has an overhead of 1–3 %. Thus, even after our integration efforts, the overheads of the reliability improvements remain low.

Overall, the transition of the original Loris, with no virtualization or reliability features, into rvLoris, which incorporates virtualization, deduplication indexing, and resilience against all four reliability threats, adds an overhead in the 6–12 % range. We believe these numbers are quite reasonable.

C. rvLoris reliability

Finally, we evaluate the reliability of rvLoris. Due to space constraints, we report on the area affected most by this work: resilience against software bugs. As a side effect, we test recovery points and thus whole-system failure recovery. Additional experiments have confirmed that our protection against memory and disk corruption has not been affected by the new changes. We do not include those results here.

We perform fault injection experiments on each of the four rvLoris layers, while running either an OpenSSH build or a version of PostMark which verifies all file data and call results. Using the framework from previous work [12], we inject two types of faults: fail-stop faults which merely crash the process (1,000 injections per configuration), and random faults which simulate the effects of common software bugs (250 injections per configuration). Once per minute, we inject 100 faults at once. Thus, in total, we inject one million faults.

We measure the number of resulting crashes in the target layer, successful recoveries, permanent failures, timeouts, crashes in other layers, and failures propagated to applications. In theory, the *fail-stop* faults should always lead to successful recovery, except in the cache layer, which may flag permanent failure due to a checksum error in the DSS; after all, DSS updates are not atomic (Sec. II-C). The *random* faults may however cause silent failures, thereby violating assumptions we made in all our works; in particular, that corrupted results are never propagated across layers. Because of the resulting risks, we perform these experiments in a virtual machine.

The results are shown in Table III. For *fail-stop* fault injection, all injections in the naming, logical, and physical layers indeed resulted in a crash and then successful, application-transparent recovery. As expected, in a small number of cases, cache-layer recovery failed on a DSS checksum mismatch. In addition, some of the cache-layer crashes caused a cross-layer memory copy failure, resulting in a crash of the logical layer due to inadequate error handling. In all other cases (98 %), the cache layer recovered successfully.

As expected, the *random* fault injections resulted in more diverse failures, including cases where no effects were observed at all. In several cases, the faults caused operations to start returning errors, thus resulting in propagation of corrupt results to other layers and often also to the application. Again, such failures are beyond the scope of our work. In many other cases, the faults caused a request or reply to be dropped, resulting in no progress (timeout); call timeouts could fix this. In the majority of cases (86 %) however, our crash recovery techniques caused the random fault injection to result in application-transparent recovery.

VI. Related Work

Our previous papers already provide overviews of work related to their respective topics. Here we discuss work related to combinations of reliability in the storage stack.

Membrane [6] implements support for checkpointing and logging for Linux file systems. Membrane can make use of file system support for recovery points, although this requires small changes to such file systems for fully correct recovery.

EnvyFS [5] uses N-version programming to protect applications from certain classes of software bugs and disk corruption, by employing multiple different file systems to perform the same actions. EnvyFS has substantial performance overheads and complicates system crash recovery.

Table III
Fault injection results, showing per layer, benchmark, and fault injection type: the number of times fault injection was performed (I), and the resulting number of target layer crashes (C), successful recoveries (R), permanent failures (P), timeouts (T), other-layer crashes (L), and application failures (A).

Layer	Benchmark	Fail-stop fault injection							Random fault injection						
		I	C	R	P	T	L	A	I	C	R	P	T	L	A
Naming	OpenSSH	1000	1000	**1000**	0	0	0	0	250	223	**216**	0	9	1	6
	PostMark	1000	1000	**1000**	0	0	0	0	250	228	**226**	0	22	0	2
Cache	OpenSSH	1000	1000	**989**	9	0	2	0	250	222	**201**	4	28	1	16
	PostMark	1000	1000	**980**	13	0	7	0	250	226	**214**	6	24	0	6
Logical	OpenSSH	1000	1000	**1000**	0	0	0	0	250	212	**212**	0	38	0	0
	PostMark	1000	1000	**1000**	0	0	0	0	250	219	**210**	0	31	1	8
Physical	OpenSSH	1000	1000	**1000**	0	0	0	0	250	222	**221**	0	28	1	0
	PostMark	1000	1000	**1000**	0	0	0	0	250	230	**228**	0	19	0	2

Z^{2}FS [19] is a variant of ZFS that can switch between checksum types to detect both memory and disk corruption at low cost, although also with lower detection guarantees.

Various generic techniques based on in-memory transaction [20] or checkpoints [18] offer the potential to recover from software bugs across the entire storage stack at once, with guarantees similar to those we provide for the cache. However, given that such techniques inherently make rollback copies for every page written to in the cache, they have exorbitant overheads when applied to the storage stack [20]. As stated in Sec. IV-E, we believe a hybrid approach could help here.

VII. Conclusion

In this paper, we have have attempted to provide a first set of answers to questions regarding the integration of several reliability techniques and other functionality in the storage stack. In the process, we have added support for virtualization to our stack. The case study has yielded mostly positive answers, as well as new areas warranting more research.

We believe that several of our findings are applicable beyond this case study. For example, our storage architecture and reliability improvements could be implemented in a monolithic environment–the latter by building on existing recovery techniques (e.g., [21]). More generally, any storage stack faces similar reliability threats, and we expect that our findings regarding the feasibility, advantages, limitations, and complexity of adding comprehensive reliability support largely apply to other storage stacks as well.

References

[1] R. Hagmann, "Reimplementing the Cedar file system using logging and group commit," in *SOSP*, 1987.

[2] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: a case for log-structured file systems," *SIGOPS Oper. Syst. Rev.*, vol. 23, pp. 11–28, 1989.

[3] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio file cache: surviving operating system crashes," in *ASPLOS*, 1996.

[4] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," in *OSDI*, 2008.

[5] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Tolerating file-system mistakes with EnvyFS," in *USENIX ATC*, 2009.

[6] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Membrane: operating system support for restartable file systems," in *FAST*, 2010.

[7] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Refuse to Crash with Re-FUSE," in *EuroSys*, 2011.

[8] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: verifying file system consistency at runtime," in *FAST*, 2012.

[9] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Loris - A Dependable, Modular File-Based Storage Stack," in *PRDC*, 2010.

[10] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Integrated System and Process Crash Recovery in the Loris Storage Stack," in *NAS*, 2012.

[11] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Battling Bad Bits with Checksums in the Loris Page Cache," in *LADC*, 2013.

[12] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Transaction-based process crash recovery of file system namespace modules," in *PRDC*, 2013.

[13] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Towards a flexible, lightweight virtualization alternative," in *SYSTOR*, 2014.

[14] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained," in *FAST*, 2008.

[15] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (Third Edition)*. Prentice Hall, 2006.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *EDCC*, 2006.

[17] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *FAST*, 2011.

[18] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Techniques for efficient in-memory checkpointing," in *HotDep*, 2013.

[19] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Zettabyte reliability with flexible end-to-end data integrity," in *MSST*, 2013.

[20] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *ASPLOS*, 2009.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP*, 2003.